15
150pts.

RUN

Enciclopedia Práctica del Spectrum

Nueva Lente/Ingelek

DATOS EN PROGRAMA

Ya hemos visto en capítulos anteriores dos de las formas posibles de asignar valores a las variables la primera de ellas, a través de la sentencia **LET**, y la segunda, mediante **INPUT**. Por ambos procedimientos, se reemplaza el valor que hasta el momento tiene una variable, del tipo que sea, para adoptar en adelante el indicado por las sentencias de asignación (**LET** o **INPUT**).

En el primer caso, el nuevo valor asignado a la variable en cuestión es el indicado por la constante o variable situada a la derecha del símbolo de igualdad. En el segundo, la asignación se realiza por el valor que introducimos mediante **INPUT**, en el mismo momento de la ejecución del programa, y es, por supuesto, el valor aceptado por el teclado.

Además de estos dos sistemas que acabamos de repasar, existe un tercero todavía desconocido para nosotros. Su objetivo es recuperar datos contenidos en el propio programa, que han sido situados en el lugar deseado, por medio de la palabra clave **DATA**.

En realidad, la operación de inclusión de datos en el programa se realiza a través de la sentencia **DATA**, mientras que la recuperación se efectúa por medio de la sentencia **READ**; pero está claro que la utilización de la segunda, implica haber hecho antes uso de la primera, por lo cual ambas instrucciones quedan estrechamente vinculadas, dado que ninguna de ellas tiene sentido sin la otra. Nos encontramos ante un nuevo caso de «simbiosis» informática, similar a la descrita en el tratamiento de bucles mediante **FOR** y **NEXT**.

Si tratamos de encontrar un sentido práctico a este conjunto de sentencias, nos damos cuenta en seguida de su utilidad para incluir datos fijos en el programa, como pueden ser nombres de datos a pedir por medio de un **INPUT**, la lista de los meses del año, etc. Así pues, dada su considerable importancia, pasemos al estudio exhaustivo de este sistema de lectura de datos.

LA SENTENCIA DATA

La palabra clave **DATA** puede incluirse en cualquier parte del programa, siendo su misión infor-

En la asignación de valores, entran en juego tres sistemas **LET, INPUT** *y* **READ-DATA-RESTORE**.

¡!

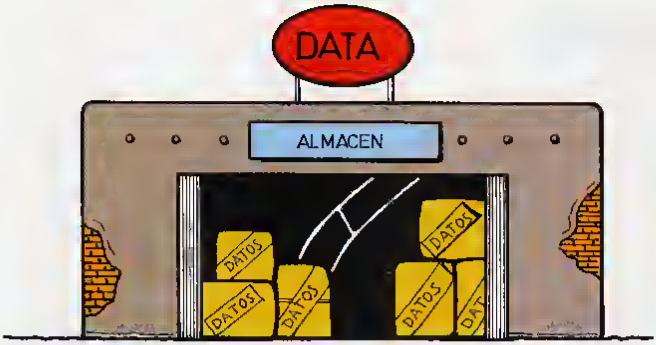
Una línea que comienza con la sentencia **DATA**, provoca una interpretación especial, calificando la línea en cuestión como datos, que serán accesibles mediante alguna sentencia **READ** situada en el programa.

*

RESTORE altera el orden secuencial de ejecución de las lecturas de los elementos de las **DATA**, de forma similar a como actúa una sentencia **GO TO** en las líneas de instrucciones convencionales.

*

Cuando tratamos de leer un elemento superior al último de los contenidos en las **DATA**, nos encontramos con el mensaje de error **E Out of DATA**.



Las sentencias **DATA** *se emplean para el almacenamiento de datos en el programa*

¡!

En una sentencia **DATA** pueden combinarse los dos tipos de datos numéricos y de cadena separados por comas (,)

*

La lectura de los elementos de las **DATA** se produce de forma secuencial a partir del numero de linea más bajo hasta el último elemento por elemento

mar al intérprete del BASIC que la línea que la contiene no debe ser interpretada como una línea de programa convencional. Su significado en Castellano es **DATOS**, y precisamente eso es lo que pretende comunicar al intérprete BASIC. «No te preocupes por esta línea, pues sólo contiene datos que serán leídos mediante **READ**».

De algún modo, existe un cierto paralelismo de concepción entre esta sentencia, y otra que ya nos es muy familiar **REM**. También en el caso de **REM**, las líneas que comienzan con esta palabra clave no son interpretadas como líneas convencionales por el ordenador, aunque existe una diferencia fundamental en cuanto a operatividad entre una sentencia y otra.

REM hace que el ordenador ignore por completo el contenido de la línea, ya que su función es meramente de ayuda a la documentación del programa. **DATA**, sin embargo, provoca una interpretación diferente, calificando la línea en cuestión como datos, que serán accesibles mediante alguna sentencia **READ** situada en el programa.

A pesar de que, como ya hemos dicho, las líneas de **DATA** pueden situarse en cualquier parte del programa, existen dos criterios preferentes de colocación, por motivos de claridad y modularidad del mismo.

El primer criterio es el de emplazar todas las líneas de **DATA** agrupadas al final del programa. El segundo, consiste en situarlas inmediatamente a continuación de la línea que contiene su sentencia **READ** correspondiente, es decir, emparejadas con la instrucción que hace uso directo de ellas.

La elección de uno u otro sistema queda al libre albedrío del programador, aunque no debemos olvidar que se trata de simples recomendaciones referentes a la forma y no al fondo, y que el programa funcionará exactamente igual si hacemos caso omiso de ellas.

Ahora bien, sin duda nos preguntaremos, que si la situación de **DATA** puede ser tan anárquica como ha podido parecer, ¿cómo sabe el ordenador qué dato leer al ejecutar un **READ**? ¿acaso puede haber un solo dato en el programa? Evidentemente, la segunda pregunta cae por su propio peso; ¡no sería de mucha utilidad poder almacenar un solo dato! Aclaremos pues la primera cuestión.

Si bien no importa el punto del programa en que se sitúen las **DATA**, sí es fundamental el orden en que se encuentren. Veremos esto más claramente con un ejemplo: supongamos que tenemos un programa de sólo dos líneas:

```
10 INPUT "NUMERO",N
20 PRINT N*N
```

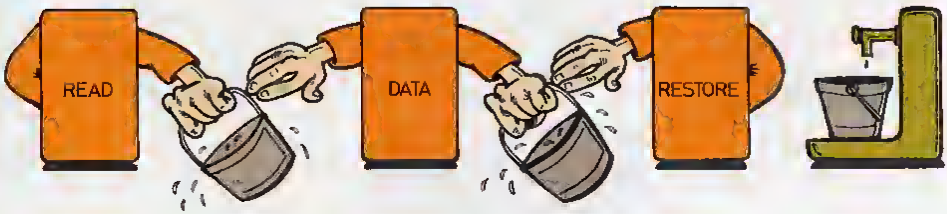
Este programa funcionará a la perfección, solicitándonos la introducción de un número, y escribiendo a continuación su cuadrado; ahora bien, ¿qué sucede si cambiamos los números de las instrucciones?

```
23 INPUT "NUMERO",N
57 PRINT N*N
```

Absolutamente nada, todo funcionará igual de bien. Sin embargo, alterar el orden de las instrucciones, aun conservando su numeración inicial, puede conllevar efectos catastróficos.

```
10 PRINT N*N
20 INPUT "NUMERO";N
```

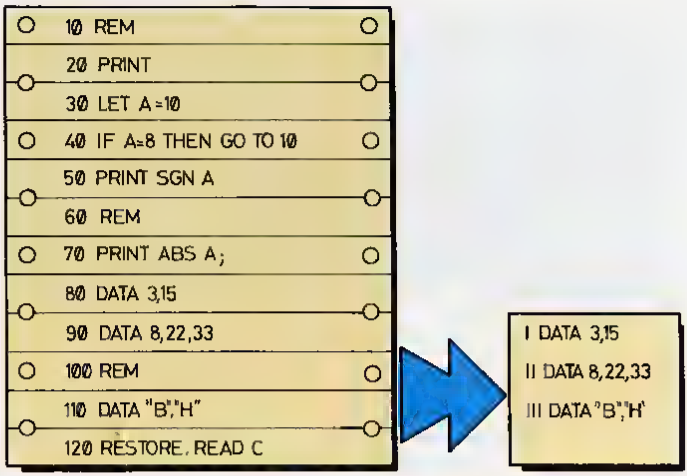
Entre **READ**, **DATA** *y* **RESTORE** *se produce un caso de «simbiosis» informática, similar al bucle* **FOR-NEXT**



De esta forma tan gráfica, habremos podido comprobar que lo importante en el programa no es el número de las líneas de instrucción, sino el orden en que estas aparezcan. Esto mismo sucede con las **DATA**.

Cuando por primera vez se ejecuta una sentencia **READ** (lectura de **DATA**), el intérprete BASIC comienza a buscar desde el principio del programa, hasta encontrar la primera sentencia **DATA**; entonces, lee el primero de los datos, y «recuerda» en qué posición se encuentra éste, mediante un elemento que denominaremos PUNTERO DE DATOS *(DATA POINTER)*. Al realizar una siguiente lectura (**READ**), el intérprete buscará el próximo dato a partir del último localizado (puntero de datos), de forma que ya no vuelva a leer otra vez el primero de los datos, almacenando la nueva posición en el puntero. Esta operación se repetirá siempre al ejecutar un **READ**.

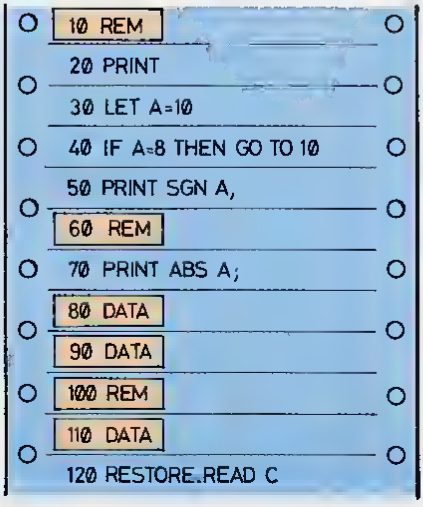
Ahora que ya conocemos el funcionamiento general del sistema de **DATA**s, entraremos de lleno en la sintaxis y características especiales de estas sentencias.



El lugar del programa en que se sitúen las sentencias DATA *es indiferente, lo que importa es el orden de colocación*

USO COMBINADO DE READ Y DATA

La primera puntualización acerca de la sentencia **DATA**, es que dentro de una misma línea pueden incluirse varios datos, separándolos con comas (,). La segunda, es que los datos no numéricos deben incluirse entrecomillados. Lo veremos más claramente en el siguiente programa de ejemplo:




```
10 REM - MESES Y DIAS
20 PRINT "M E S E S","DIAS"
30 FOR I=1 TO 12:READ X$:PRINT AT
I+1,0;X$:NEXT I
40 FOR I=1 TO 12:READ X:PRINT AT
I+1,16;X:NEXT I
50 DATA "ENERO","FEBRERO","MAR
ZO","ABRIL"
60 DATA "MAYO","JUNIO","JULIO","A
GOSTO"
70 DATA "SEPTIEMBRE","OCTU
BRE","NOVIEMBRE","DICIEMBRE"
80 DATA 31,28,31,30,31,30,31,31,30,31,
30,31
```

Como breve comentario a este programa, podemos decir que, en la línea 20, se imprime la cabecera de las dos columnas de datos, en la línea 30 se sitúa el bucle de lectura e impresión de los nombres de los meses del año y, por último, en

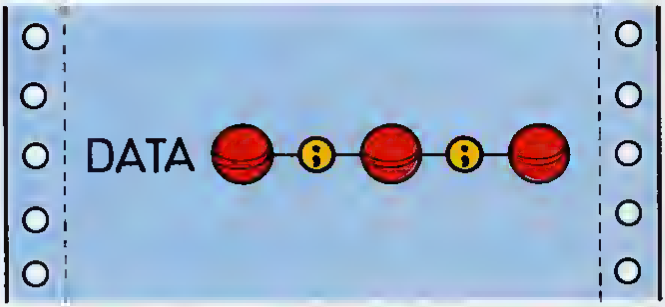
Las sentencias DATA, *al igual que* REM, *son ignoradas por el intérprete BASIC al pasar por ellas*

¡!

RESTORE puede ir seguida de un argumento numérico, que indica el número de línea donde se quiere recomenzar la lectura de datos mediante **READ**, de carecer de este argumento, se produce una restauración total al primer elemento de la primera línea de **DATA**.

*

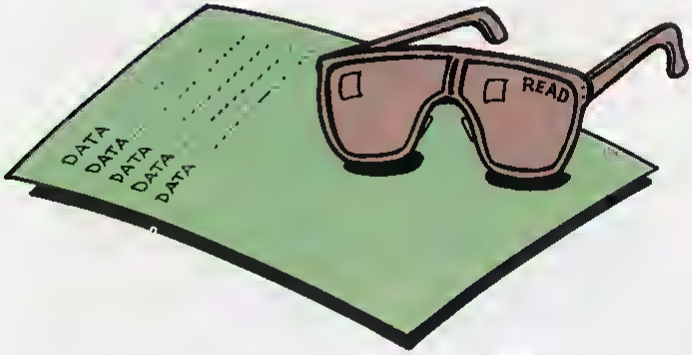
En la lectura de **DATA** hemos de tener cuidado para que el tipo de dato leído corresponda con el de la variable a asignar.



Dentro de una misma línea DATA, *se pueden incluir varios elementos separados por comas* (,)

la línea 40, se hace lo propio con el número de días de cada mes.
Hemos dejado deliberadamente en el olvido las líneas 50 a 80, que contienen los datos (**DATA**) para las sentencias **READ.**

La sentencia READ *se utiliza para la lectura de* DATAs.



Las sentencias DATA *pueden cobijar tanto datos numéricos, como de cadena, o combinaciones de ambos.*

En ellas debemos significar dos cosas: la primera, la calificación del tipo de dato a leer por medio de las comillas, y la segunda, el hecho de que el intérprete, en su discurrir secuencial por las líneas de programa, ha pasado por encima de las **DATA** sin producirse ningún tipo de inconveniente.
Este hecho se debe a que, como hemos dicho anteriormente, son calificadas como líneas sólo utilizables por las sentencias **READ** del programa, en este caso, las de las líneas 30 y 40. Como comprobación, podemos incluir una sentencia **STOP** en la línea 45, obteniendo idénticos resultados.

```
10 REM  MESES Y DIAS
20 PRINT "M E S E S","DIAS"
30 FOR I=1 TO 12:READ XS:PRINT AT
I+1,0;XS:NEXT I
40 FOR I=1 TO 12:READ X:PRINT AT
I+1,16;X:NEXT I
45 STOP
50 DATA "ENERO","FEBRERO","MAR
ZO","ABRIL"
60 DATA "MAYO","JUNIO","JULIO","A
GOSTO"
70 DATA "SEPTIEMBRE","OCTU
BRE","NOVIEMBRE","DICIEMBRE"
80 DATA 31,28,31,30,31,30,31,31,30,31,
30,31
```

Como también habremos notado, el tipo de variable que se utiliza para **READ**, debe corresponder con el tipo de dato leído de la **DATA**, puesto que de no ser así, el programa se detendría con un error **C Nonsense in BASIC.** Así pues, recapitulando, el proceso completo que se desencadena durante la ejecución de un **READ** es
1 Lectura del próximo elemento de **DATA.**
2 Actualización del puntero de datos y
3 Asignación del dato leído a la variable indicada tras **READ.**
Otra característica interesante de la sentencia **DATA**, es la de poder mezclar los dos tipos de datos, numéricos y de cadena, en una misma línea. Esto nos va a permitir codificar de nuevo el pro-

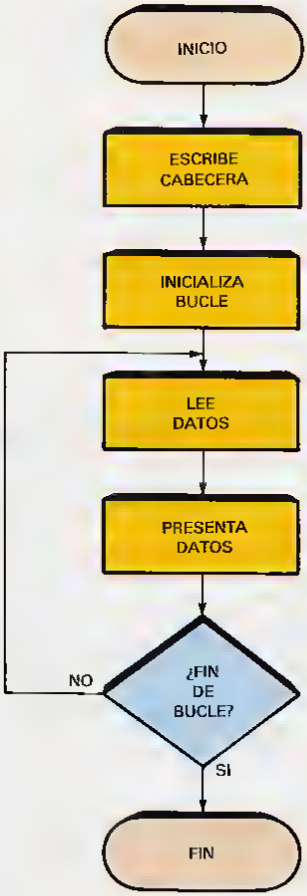


Diagrama de flujo del programa "MESES Y DIAS"

grama, de una forma mas simple, para escribir los datos en horizontal, y no en vertical, como en el programa del ejemplo anterior

```
10 REM - MESES Y DIAS
20 PRINT "M E S E S","DIAS""
30 FOR I=1 TO 12.READ X$,X.PRINT X$,X.NEXT I
40 DATA "ENERO",31,"FEBRERO",28,"MARZO",31
50 OATA "ABRIL",30,"MAYO",31,"JUNIO",30
60 DATA "JULIO",31,"AGOSTO",31,"SEPTIEMBRE",3D
70 DATA "OCTUBRE",31,"NOVIEMBRE",3D,"DICIEMBRE",31
```

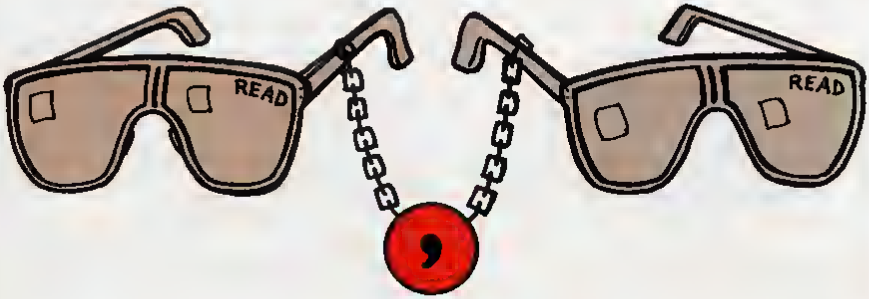
El nuevo programa contiene además una modificación sustancial, que ha eliminado la antigua línea 40 en la linea 30 podemos apreciar una sentencia **READ** con sus elementos separados por comas Del mismo modo que la sentencia **INPUT** permite la petición de más de un dato, separando éstos por una coma (,), puede efectuarse la lectura de varios elementos por medio de **READ**, separando los elementos también mediante coma (,).

Antes de continuar, conviene hacer una aclaración acerca del comportamiento de **READ** con **OATA**s de más de un elemento Realmente este es el caso más frecuente, y **READ** los trata como si estuvieran en líneas **OATA** separadas, es decir, el puntero de datos no señala la última línea leída mediante **READ**, sino el último dato leído Debido a esto, la estructura siguiente **10 DATA 31,28,3D,31**, es totalmente equivalente a

```
1D DATA 31
20 OATA 28
30 DATA 30
40 DATA 31
```

De igual modo, el tratamiento de **READ** con separación de comas, es el mismo que si se encon-

Varios **READ**, *al igual que* **DATA**, *se pueden encadenar mediante comas* (,)



¡!

Pueden incluirse varios elementos dentro de una misma linea de **DATA**, separándolos con comas (,)

*

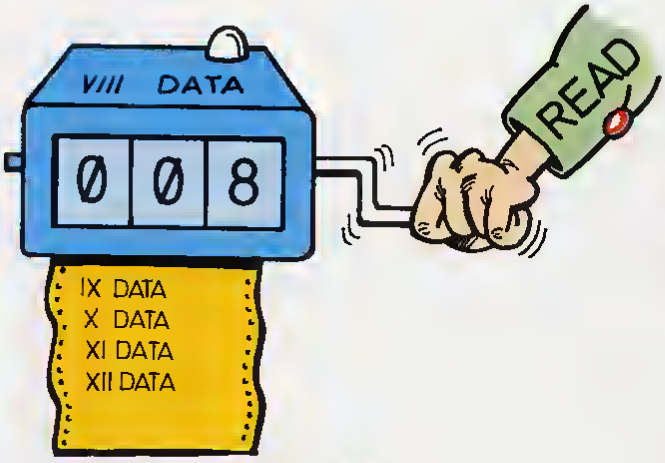
Por medio del conjunto de instrucciones **READ** y **DATA** pueden recuperarse datos contenidos en el propio programa Ambas sentencias quedan estrechamente vinculadas puesto que ninguna de ellas tiene sentido sin la otra

*

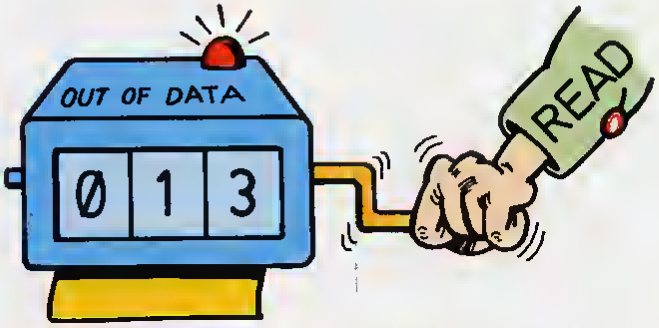
La palabra clave **DATA** puede incluirse en cualquier parte del programa, siendo su misión la de informar al interprete del BASIC, que la linea que la contiene no debe ser interpretada como una linea de programa convencional

*

Varias sentencias **READ** pueden ser encadenadas mediante comas (,)

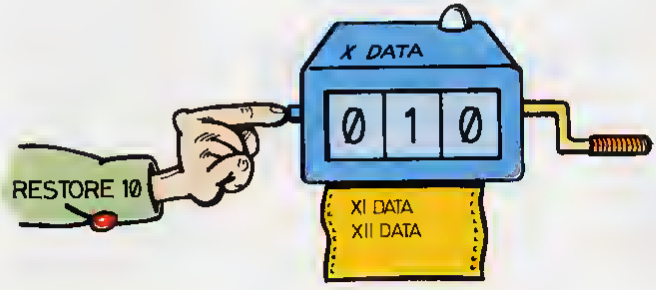


Cada vez que se ejecuta un **READ**, *se actualiza el puntero de datos*



Al intentar efectuar más READs *que* DATAs *tiene el programa, produciremos un error del tipo* E Out of DATA.

El RESTORE *puede operar a partir del número de línea que deseemos*



traran en líneas diferentes el dato leído de **DATA** es asignado exclusivamente a la variable que corresponda. Por tanto, la línea **10 READ X,Y**, tiene un efecto idéntico a...

```
10 READ X
20 READ Y
```

LA SENTENCIA RESTORE

Según lo dicho hasta ahora, de forma similar a como el intérprete del BASIC evalúa el programa, instrucción por instrucción, a partir del número de línea más bajo hasta el último, se produce la lectura de los elementos de las **DATA**. Cuando el programa llega a su término, por carecer de una siguiente instrucción a ejecutar, este se detiene. Sin embargo, ¿qué sucederá si agotamos todas las **DATA** de un programa y ensayamos un nuevo **READ**?

Obtendremos, lógicamente, un mensaje de error, en este caso del tipo **E Dut of DATA** (Fuera de **DATA**). Lo cierto es que, en algunas ocasiones, y debido a necesidades de programación, se precisa leer una y otra vez mediante **READ** las mismas líneas **DATA**, ello implica la necesidad de volver atrás el puntero de datos, algo parecido a «desleer» los elementos leídos por **READ**.

Para solucionar este problema, el BASIC dispone de la sentencia **RESTDRE** (RESTAURA). Esta altera el orden secuencial de ejecución de lecturas, dentro de los elementos de **DATA**, de forma similar a la alteración que en el discurrir de un programa, produce una sentencia **GD TD**. Para ser más exactos, tiene la propiedad de situar el puntero de datos al comienzo del programa, de igual modo que si acabara de ser ejecutado mediante **RUN**.

RESTDRE va aun más lejos, y nos permite colocar el puntero de datos en el número de línea que deseemos, si bien no a partir del dato concreto que queramos. La sentencia **RESTDRE** puede ir seguida o no de un argumento numérico, correspondiente al numero de línea donde se desea recomenzar la lectura de datos mediante **READ**; de carecer de este argumento, el BASIC supone que la restauración de las **DATA** debe ser total, debido a lo cual, se le otorga al puntero el valor de la primera línea.

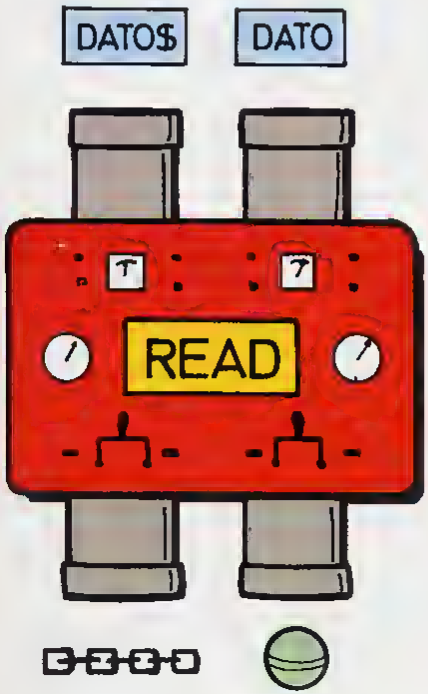
Como en el caso de la situación de las **DATA**, carece de importancia que el **RESTDRE** se ejecute a una línea donde no exista ningún dato. Simplemente, el ordenador considerará ese punto como

inicio de la busqueda de una **DATA** a leer **(READ)**.

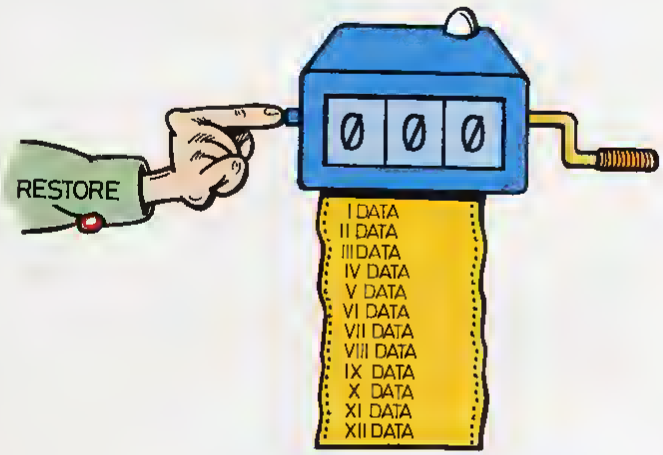
Veremos ahora un ejemplo de utilización conjunta de las sentencias **READ, DATA,** y **RESTORE,** empleadas para la depuración de errores, en la introducción de una fecha por el teclado

```
  10 REM DEPURACION DE FECHAS - J M.LOPE
Z MARTINEZ
  20 INPUT 'Fecha:'; LINE F$: IF LEN F$<
>6 THEN  GO TO 110
  30 FOR I=1 TO 6
  40 IF F$(I)<'0' OR F$(I)>'9' THEN  GO
TO 110
  50 NEXT I
  60 IF NOT VAL F$( TO 2) OR NOT VAL F$(
3 TO 4) OR VAL F$(3 TO 4)>12 THEN  GO TO
 110
  70 RESTORE
  80 FOR I=1 TO VAL F$(3 TO 4): READ D:
NEXT I
  90 IF VAL F$( TO 2)>D THEN  GO TO 110
 100 LET E$='': GO TO 120
 110 LET E$='ERROR!'
 120 PRINT F$,E$: GO TO 20
 130 DATA 31,29,31,30,31,30,31,31,30,31,
30,31
```

Al efectuar **READ**, *hay que vigilar que el tipo de variable a asignar coincida con el tipo de dato a leer*



Este pequeño programa resuelve, aunque no de forma exhaustiva, el problema de depurar la entrada de una fecha que deseamos sea coherente, imprimiendo por cada nueva entrada dos columnas, en la primera de las cuales figura el dato introducido mediante la sentencia **INPUT**, y en la segunda un indicativo de si éste es o no correcto. Decimos que la depuración no es exhaustiva, por



La sentencia **RESTORE**, *se emplea para restaurar a su posición inicial el puntero de datos*

que no tiene en cuenta para nada el año, es decir, admite desde el 00 hasta el 99, no reparando por tanto en si éste es bisiesto o no, consecuentemente, siempre se admite la fecha del 29 de febrero. Para que podamos comprobar el funcionamiento del programa, deberemos introducir la fecha en el formato dia (2 dígitos), mes (2 dígitos) y año (2 dígitos). Así por ejemplo, la fecha 15 de julio de 1908, se expresaría como 150708

Dicho esto, entraremos en el comentario del programa. En la línea 20, se acepta la fecha **F$** desde el teclado, mediante un **INPUT LINE**. Recordaremos que esta sentencia tiene la ventaja de no efectuar la petición del dato alfanumérico entre comillas, y que para interrumpir el programa durante su ejecución, debemos teclear **CAPS SHIFT + 6**. En la propia línea 20, se establece un control para averiguar si la longitud de la cadena de caracteres entrada es o no diferente de 6. Si lo es, se salta a la instrucción 110, donde se canaliza la gestión de fechas con error.

En las líneas 30 a 50 se comprueba si alguno de los seis caracteres entrados no fuera numérico. Esto se consigue con un bucle **FOR-NEXT** desde 1 hasta 6, comprobando carácter a carácter si és

¡!

Las líneas de **DATA** pueden situarse en cualquier parte del programa, aunque suele seguirse el criterio de incluirlas al final del mismo, o inmediatamente a continuación de la sentencia **READ** que hace uso de ellas

*

En las sentencias **DATA** los datos no numéricos deben incluirse entrecomillados

tos son inferiores a 0 o superiores a 9. Caso de serlo cualquiera de ellos, se salta a la instrucción 110, de no ser así, el programa continúa en secuencia con las siguientes instrucciones.

Ya nos es conocida la potencia de interpretación de la sentencia **VAL**, y es debido a esto que nos vemos obligados a la inclusión en el programa de las líneas 30 a 50. El motivo es que, en las líneas de programa siguientes, hacemos comprobaciones con los valores numéricos (**VAL**) de los tres segmentos de dos caracteres, de los cuales se compone la cadena **F$**; donde los dos primeros caracteres de la cadena indican el día, el tercero y el cuarto el mes, y los dos últimos (quinto y sexto), el año.

Si hubiéramos permitido que se filtrara algún carácter no numérico, por ejemplo **A4**, en las posiciones tercera y cuarta de la cadena, correspondientes al mes, al calcular el **VAL** de este segmento de la cadena, hubiéramos obtenido un error **2 Variable not found** (variable no encontrada), puesto que el intérprete BASIC supone que, al no ser numérico el segmento analizado, debe buscar el valor de la variable **A4** en la memoria del ordenador, lo cual motiva la aparición del error.

En la línea 60 se establece la comprobación de si el día y el mes son cero y, en el caso del mes, si además no es superior a 12. Al detectarse cualquiera de estos errores, puesto que los operadores relacionales son de tipo **OR**, se bifurca la ejecución a la instrucción 110.

En la línea 70 se ejecuta una sentencia **RESTORE**. Esta, al carecer de número de instrucción como argumento, inicializa el puntero de lectura de **DATA** a la primera línea. Por otra parte, en la línea 80, hemos codificado un bucle **FOR-NEXT**, que efectúa tantas lecturas **READ** de la **DATA**, como indica el **VAL** del segundo segmento de la cadena **F$**, que contiene los meses.

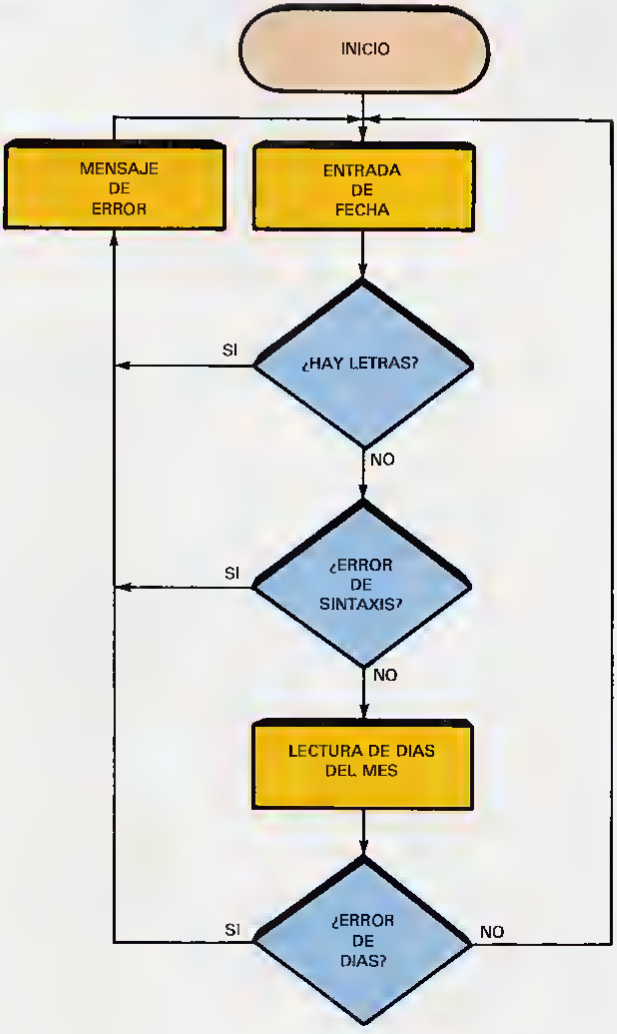
En la línea 90, y una vez obtenido el valor de la variable **D**, que indica el número de días de que consta el mes, se efectúa la comprobación de si los días, primer segmento de la cadena **F$**, son o no superiores a este número máximo. Caso de serlo, el programa se desvía a la instrucción 110.

Si todas las comprobaciones de error hechas hasta el momento han fracasado, el programa continúa en secuencia y accede a la línea 100, en la cual se anula el contenido de la variable **E$**, y se salta a la instrucción 120, para imprimir en la pantalla la cadena tecleada y el contenido de la variable **E$**, que puede ser nada o el mensaje "ERROR!".

Para el caso de los saltos que se puedan producir a la línea 10, por detección de errores en la depuración, se asigna el valor de "ERROR!" a la variable **E$** y, seguidamente, se pasa a la impresión en dos columnas de la línea 120, para volver inmediatamente al **INPUT** de la 20, cerrando el ciclo de programa.

Debemos observar, para valorar en su medida la utilidad de la sentencia **RESTORE**, que el programa vuelve al **INPUT** sin cesar, tanto en el caso de que se hayan producido errores como si no. De no haber incluido la instrucción 70 que contiene el **RESTORE**, la primera pasada del programa hubiera resultado satisfactoria, pero la segunda habría producido un error, al intentar leer elementos por **READ** más allá de los incluidos en la instrucción 130.

Diagrama de flujo del programa "DEPURACION DE FECHAS".



GRAFICOS ANIMADOS

El último capítulo nos sirvió de introducción al dinamismo a pantalla completa, y tuvimos la oportunidad de comprobar la eficacia de una de las técnicas a este fin: el dinamismo por intermitencia. Sin embargo, quedó pendiente el estudio de otras dos técnicas, a cual más importante, que concentraban su acción independientemente sobre el área de atributos, y sobre el archivo de imagen. Pasaremos a continuación a dar cumplida información de ambas.

CONTROL DE ATRIBUTOS

Como ya sabemos, la imagen compuesta en la pantalla del televisor por nuestro ordenador, no es a efectos de programación un todo único, ésta se divide en dos componentes absolutamente diferentes: la forma de la pantalla y el color de la misma. La técnica de dinamismo por atributos, se basa precisamente en la animación de uno solo de estos componentes: el color.

Efectivamente, la alteración rápida de los colores de la pantalla, aun sin afectar a la forma de la misma, es decir, los gráficos, textos, etc , que la integran, tiene de por sí un efecto dinámico de gran interés. Vamos a comprobarlo mediante un sencillísimo ejemplo:

```
10 INK 0
20 PRINT AT 11,12;"MENSAJE"
30 PAUSE 5
40 INK 7
50 PRINT AT 11,12,"MENSAJE"
60 PAUSE 5
70 GO TO 10
```

Aparentemente, el contenido escrito en el archivo de imagen se altera, apareciendo y desapareciendo, sin embargo, tanto en la línea 30 como en la 50 escribimos un idéntico mensaje. Efectivamente, el contenido del archivo de imagen no ha variado, no obstante, hemos alterado el del área de atributos, cambiando el color de escritura del mensaje, alternando la tinta 0 (negro), con la 7 (blanco, que sobre fondo blanco se hace imperceptible).

Sería por tanto interesante poder someter a toda la pantalla, o al menos a gran parte de ella, a un proceso similar. Dado que las dimensiones del área de atributos son de relativa consideración (768 bytes), y que precisamos un efecto prácticamente simultáneo sobre todo el área a tratar, nos veremos en la obligación de recurrir al código máquina.

La subrutina que presentamos a continuación, denominada **ATRIBUTOS**, somete a cada byte del área del mismo nombre, a las siguientes transformaciones:

* Incrementa el código de la tinta, cuidando de no sobrepasar el valor 7.
* Incrementa el código de fondo, teniendo la misma precaución que en el paso anterior.
* Deja inalteradas las cualidades de **BRIGHT** (brillo) y **FLASH** (intermitencia).

Todo esto puede que no nos parezca de un excesivo interés, sin embargo, merece la pena que introduzcamos el siguiente programa de demostración, su efecto sin duda nos sorprenderá.

```
 10 REM ATRIBUTOS - C.DE LA OSSA & F.LOPEZ MARTINEZ
 20 DATA 1,0,80,17,0,3,10,60,230,7
 30 DATA 103,10,198,8,230,56,111,10,230,192
 40 DATA 133,132,2,3,27,122,179,32,233,201
 50 CLEAR 32499: FOR I=0 TO 29
 60 READ J
 70 POKE 32500+I,J
 80 NEXT I
 90 BORDER 1: PAPER 1: CLS
100 FOR I=0 TO 7
110 PRINT  PAPER I;'
'
120 NEXT I
130 PRINT AT 2,14; PAPER 2; INK 9;"RUN";AT 3,5; PAPE
R 3;"ENCICLOPEDIA PRACTICA";AT 4,9; PAPER 4;"DEL  SPE
CTRUM"
140 FOR I=0 TO 19
150 READ J,K: LET L=I-8*INT (I/8)
160 PRINT AT J,K; PAPER L;"    ";AT J+1,K; PAPER L;"
  ";AT J+2,K;'
170 NEXT I
180 DATA 9,15,10,13,11,11,12,9,13,7,14,5
190 DATA 15,7,16,9,17,11,18,13,19,15
200 DATA 18,17,17,19,16,21,15,23,14,25
210 DATA 13,23,12,21,11,19,10,17
220 FOR I=0 TO 10: CIRCLE  INK 7;27,11,I: NEXT I
230 FOR I=0 TO 10: CIRCLE  INK 7;220,11,T. NEXT I
240 FOR I=0 TO 10: CIRCLE  INK 7;27,88,T: NEXT I
250 FOR I=0 TO 10: CIRCLE  INK 7;220,88,I. NEXT I
260 PRINT AT 15,12;"ATRIBUTOS"
270 LET I=5
280 RANDOMIZE USR 32500
290 LET I=I+1: IF I=8 THEN  LET T=0
300 BORDER T
310 PAUSE 5
320 GO TO 200
```

¡!

Al efectuar **POKE**s al área de atributos debemos de tener mucho cuidado para no exceder su límite inferior puesto que a continuación se encuentra el programa en **BASIC** y su deterioro puede ser irremediable.

BITS

Una forma muy rápida de traducir una palabra a su forma bajo alto es efectuar **RANDOMIZE <número a codificar>**. Acto seguido el **PEEK** de la dirección 23670 contendrá su peso bajo y **PEEK** de 23671, el alto. He aquí un programa de utilidad basado en esta facultad

```
10 REM BAJO ALTO
20 INPUT "NUMERO ? ";N
30 IF N<0 OR N>65535 THEN GO TO 20
40 RANDOMIZE N
50 PRINT "BAJO ";PEEK 23670
60 PRINT "ALTO ";PEEK 23671
```

*

Las posiciones investigadas corresponden a la variable del sistema **SEED**, cuya misión es servir de «semilla» en la extracción de números aleatorios

*

Para proteger el código máquina de la sobreescritura por el BASIC utilizamos la sentencia **CLEAR** seguida de un argumento numérico, con el cual se señala la última dirección utilizable por el BASIC. Este sistema presenta el gran inconveniente de borrar también las variables. Para proteger el código máquina, sin necesidad de borrar las variables, podemos acceder directamente a la variable del sistema **RAMTOP**. **RANDOMIZE <valor de tope de memoria>: POKE 23730,PEEK 23670: POKE 23731,PEEK 23671**

Ahora que ya hemos salido de nuestro asombro y somos conscientes de la gran utilidad de la subrutina de atributos, podemos pasar a un estudio más detallado de la misma. Comencemos por el listado de ensamblador de la página siguiente. Esencialmente, se utilizan dos pares de registros el BC y el DE. El primero de ellos sirve de puntero del byte a tratar en el área de atributos, y por tanto comienza siendo 22528 (primera posición de dicha zona), que va incrementándose según se va completando el tratamiento de los bytes. El par DE se utiliza como contador del número de bytes a analizar, y comienza siendo 768 (total de bytes del área de atributos), valor que se va decrementando conforme se van alterando los bytes; cuando DE llega a cero, la subrutina ha terminado su trabajo.

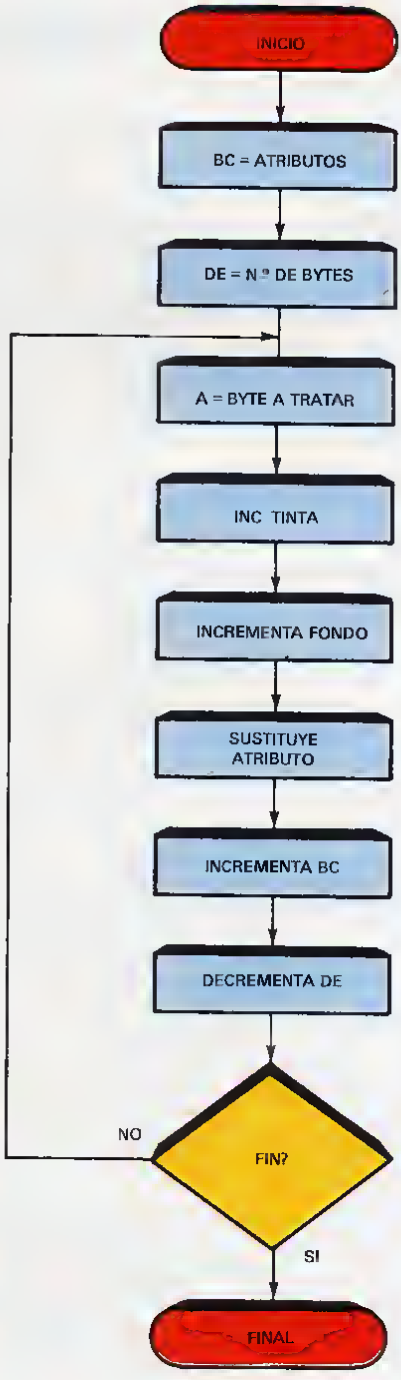
Quizás pensemos que la utilización de DE podía haber sido evitada, simplemente utilizando también BC para investigar si se ha llegado al final de la pantalla (23295), sin embargo, utilizar DE concederá una mayor flexibilidad a la rutina. Veamos por qué.

Si deseamos tratar todos los bytes de la pantalla, la subrutina no precisará ningún cambio, sin embargo, con su actual programación, es enormemente sencillo alterar el número de bytes a tratar (sólo con hacer los **POKE**s correspondientes en bajo-alto, al quinto y sexto byte de la rutina). Del mismo modo, cambiar el primer byte a analizar, también será muy fácil, puesto que sólo tendremos que depositar la primera dirección a tratar (en bajo-alto), en el segundo y tercer byte de la rutina. Si llevamos a cabo estas modificaciones, hemos de tener cuidado para que el byte inicial más el número de bytes alterados no sobrepase el último byte de la zona de atributos (23295).

Por otra parte, todos los que tengamos unos mínimos conocimientos de código máquina, sabremos que incrementar o decrementar pares de registros es igualmente sencillo, sin embargo, a la hora de las comprobaciones, es mucho más fácil investigar si un par de registros es cero (un simple OR entre los registros que lo componen), que si ha alcanzado determinado valor.

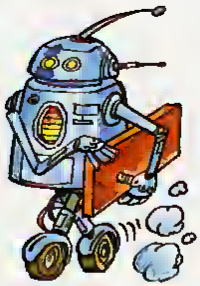
Para la adopción de la rutina por nuestra propia cuenta, podemos utilizar el siguiente programa

```
10 REM RUTINA DE ATRIBUTOS - © DE LA OSSA & F. LOPEZ MARTINEZ
20 CLEAR 32499
30 DATA 1
40 DATA 0,88
50 DATA 17
60 DATA 0,3
70 DATA 10,60,230,7,103,10,198,8,230,56,111,10,230,192,133,132,2,3,27,122,179,32,233,201
```



SUBRUTINA DE ATRIBUTOS

ETIQUETA	OBJETO	FUENTE	COMENTARIO
ATTR	1 0 88	LD BC,22528	,Carga HL con la primera dirección a ,tratar
	17 0 3	LD DE,768	,Carga DE con el numero de bytes a tratar
LOOP	10	LD A,(BC)	,Inicia el ciclo de tratamiento car gando el ,acumulador con el byte a tratar.
	60	INC A	,Incrementa de uno A, y por tanto el có ,digo de tinta
	230 7	AND A,7	,Pone a cero los bits del 3 al 7, y deja inal ;terados los restantes (del 0 al 2). Así ,pues, si la tinta pasó a ocho, queda a cero ,y si no, con el incremento correspon ;diente
	103	LD H,A	,Guarda el estado actual del acumulador
	10	LD A (BC)	,(nuevo codigo de tinta) en H ;Carga el acumulador con el byte a anali- ;zar. Por tanto, el valor anterior se pierde, ,por lo cual fue guardado previamante ,en H
	198 8	ADD A,8	,Suma 8 al acumulador, es decir, incre- ,menta de uno el código de fondo
	230 56	AND 56	,Pone a cero los bits 0, 1, 2, 6 y 7, y deja ,inalterados los restantes (del 3 al 5) Así ,pues, si el código de fondo paso a 8 que ,da a cero, si no, mantieel incremento
	111	LD L,A	,Guarda el acumulador en L, es de cir, el ,código de fondo
	10	LD A,(BC)	,Carga el acumulador con el byte a ana- ,lizar, de ahi que al valor anterior fuera ,preservado en L
	230 192	AND 192	,Pone a cero los bits del 0 al 5 y deja inal ,terados los restantes (6 y 7). Gracias a ,esto, anula los bits de tinta y fondo, tra- ,tados anteriormente, y deja en el acumu- ,lador sólo los de brillo e intermitancia, ,que no serán modificados
	133	ADD A,L	,Suma L al acumulador, as decir, añade a ,brillo e intermitencia ante riores, el nue ,vo código de fondo creado por la su- ,brutina
	132	ADD A,H	,Suma H al acumulador, por tanto, añade ;al A anterior al codigo de tinta
	2	LD (BC),A	,Deposita el nuevo atributo en la posición ,correspondiente
	3	INC BC	,Incrementa de uno la posición a analizar
	27	DEC DE	,Decrementa de uno el numero de bytes ,que quedan por tratar
	122	LD A D	,Carga el acumulador con D, preparándo- ,lo para la comprobación de fin de taraa
	179	OR E	,Suma lógica de D y E Su resultado será ,comprobado en la siguiente instrucción
JUMP	32 233	JR NZ LOOP	,Si D OR E distinto de cero, DE es distinto ,de cero, por lo cual volve mos al comien- ,zo del ciclo de tra tamiento
END	201	RET	,Si D OR E es cero implica qua DE=0, por ,tanto, no quedan bytes por tra tar, y po- ,damos devolver el control al BASIC



¿?

Con la expresion QUE DAR COLGADO, se designa usualmente la pérdida de control sobre el ordenador

*

En la lectura de listados, el símbolo # suele denominarse almohada *number* (pronunciado *namber*) o *hush* (pronunciado *hash*)

*

En la lectura de listados los elementos seguidos de un argumento entre paréntesis como las variables suscritas o las partes de cadenas se suelen leer como <elemento> DE <argumento entre parentesis> Así por ejemplo **G (H,L)**, se lee, "G DE H COMA L"

```
80 FOR I=0 TO 29
90 READ A
100 POKE 32500+I,A
110 NEXT I
```

Según el lugar donde deseemos situar la subrutina, deberemos sustituir el **POKE 32500+I,A** de la línea 100, y por supuesto, el **CLEAR** de protección de la rutina, establecido en la línea 20, que siempre debe estar al menos una posición antes que la de comienzo de la rutina. Una vez ejecutado este programa, el código máquina quedará almacenado en la memoria, y podrá ser utilizado en cualquier momento mediante **RANOOMIZE USR** <dirección de carga> (en el ejemplo, **RANOOMIZE USR 32500**).

Para finalizar la adaptación de la rutina a nuestra propia conveniencia, los bytes de la **OATA** en la línea 50, corresponden, en peso bajo-peso alto, al primer byte a tratar en el área de atributos, inicialmente 22528 (0+88*256=22528). Por otra parte, la **OATA** de la línea 60 contiene el número de bytes a analizar, expresador por el mismo sistema (bajo-alto), que inicialmente especifican 768 (3*256=768).

Así por ejemplo, si deseamos que sólo estén afectados por la rutina de atributos los últimos dos tercios de la pantalla, deberemos sustituir la línea 20 del programa de demostración por **20 OATA 1,0,89,17,0,2,10,60,230,7**. Puesto que el primer byte a tratar (primero del segundo tercio) es 22784 (22784=0+ 256*89), y el área son 16 líneas de 32 caracteres, es decir, 512 bytes (512=0+256*2).

Por último, y para comprender completamente el trabajo realizado por la subrutina de atributos, podemos diseñar una subrutina que realice la misma funcion en lenguaje **8ASIC**, la lentitud será muy grande, pero eso ayudara a que podamos apreciar perfectamente todos los cambios que se operan. Para utilizar esta subrutina propuesta, sólo hemos de añadirla al programa de demostración, y sustituir la llamada al código máquina de la línea 280, (**RANDOMIZE USR 32500**), por el acceso a la subrutina **280 GOSUB 1000.**

```
1000 LET BC=22528
1010 LET DE=64
1020 LET A=PEEK BC
1030 LET A=A+1
1040 LET M=7: GO SUB 1200
1050 LET H=A
1060 LET A=PEEK BC
1070 LET A=A+8
1080 LET M=56: GO SUB 1200
```

```
1090 LET L=A
1100 LET A=PEEK BC
1110 LET M=192. GO SUB 1200
1120 LET A=A+L
1130 LET A=A+H
1140 POKE BC,A
1150 LET BC=BC+1
1160 LET DE=DE-1
1170 IF DE<>0 THEN  GO TO 1020
1180 RETURN
1190 REM OPERACION LOGICA A AND M
1200 LET B=A: GO SUB 1310: LET A$=B$
1210 LET B=M: GO SUB 1310: LET M$=B$
1220 FOR J=1 TO 8
1230 LET A$(J)=STR$ (VAL A$(J)*VAL M$(J)
)
1240 NEXT J
1250 LET A=0
1260 FOR J=1 TO 8
1270 LET A=A+VAL A$(J)*2^(8-J)
1280 NEXT J
1290 RETURN
1300 REM DECIMAL => BINARIO
1310 LET B$=""
1320 IF B<2 THEN  GO TO 1360
1330 LET C=INT (B/2): LET R=B-C*2
1340 LET B$=STR$ R+B$
1350 LET B=C: GO TO 1320
1360 LET B$="0000000"+STR$ B+B$
1370 LET B$=B$(LEN B$-7 TO )
1380 RETURN
```

PESO BAJO - PESO ALTO

Como ya sabemos, la unidad de memoria más ampliamente difundida es el BYTE, compuesto por ocho bits, y en base al cual se mide la capacidad de memoria de un ordenador. Puesto que un bit puede portar dos informaciones distintas (0 ó 1), un byte puede codificar hasta 256 datos diferentes (desde 0 hasta 255). No obstante, en algunas ocasiones, esta cantidad de información es insuficiente, por lo cual tenemos que utilizar alguna unidad superior de información.

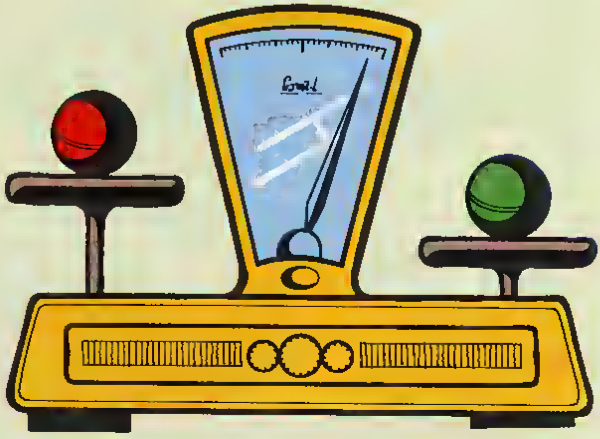
Esta es, la PALABRA (en inglés *WORD*), que se entiende compuesta por dos bytes, y por ende, por 16 bits. Así pues, la palabra puede portar 65536 informaciones diferentes. Es por ello, por ejemplo, que nuestro ordenador puede llegar a tener en el total de su memoria, precisamente 65536 bytes (64 K), y esto no ha sido por capricho, sino porque esta es la máxima cantidad de información representable en una palabra (dos bytes).

Ahora bien, la forma de leer un byte de la memoria, o depositarlo en ella, es bien sencilla, gracias a que disponemos de sentencias BASIC especializadas en ello: **PEEK** y **POKE**. Sin embargo, ¿cómo hacer para trabajar con palabras? Primero es necesario que tengamos claro un concepto acerca del «peso» de los dígitos en una cifra. Si tomamos el número decimal 23 y le damos la vuelta, conseguiremos un valor completamente diferente (32), pese a que hemos empleado los mismos dos dígitos. Esto se debe a que el valor de un número, no sólo depende de sus dígitos, sino también de la posición de éstos en la cifra, cuanto más a la izquierda se encuentre el dígito, más influye en la magnitud del número. Se dice entonces, que los dígitos de más a la izquierda son más SIGNIFICATIVOS, o también, que tienen mayor PESO.

Con los dos bytes que configuran una palabra, sucede exactamente lo mismo: no es igual la palabra 34 - 201, que la 201 - 34. Es importante que se nos indique por tanto, cual de los dos bytes es el más significativo. Lo más habitual es que las palabras se expresen en forma de PESO BAJO - PESO ALTO, por tanto, siempre debemos suponer que se encuentran en este formato, salvo que se haga una especificación al contrario.

Una vez sabido cuál es el peso bajo y cual el alto, la decodificación de la información que contiene la palabra es bien simple: el peso bajo, mas el producto del peso alto por 256. Así por ejemplo, la palabra 34 - 201 (bajo-alto), equivale a 34+256*201=51490.

La lectura de una palabra es justo la operación inversa; el peso alto se obtendrá como la parte entera del número a codificar dividido entre 256, y el peso bajo, como dicho número menos el producto de 256 por el peso alto. Siguiendo el mismo ejemplo, el peso alto de 51490 es la parte entera (cociente) de 51490/256=201, y el peso bajo, 51490-256*201=34. Esta operación realizada en BASIC sería **ALTO=INT (NUMERO/2S6): BAJO=NUMERO-256*ALTO.**



ACORAZADO

A bordo de un poderoso acorazado, la tensión se hace casi insoportable al contemplar la tripulación como dos mortíferas estelas se aproximan peligrosamente. Con una hábil maniobra, el buque consigue evitar el alcance y lanza cuatro cargas de profundidad encaminadas a la destrucción de su escurridizo enemigo.

Esta, que bien pudiera ser una narración ambientada en la Segunda Guerra Mundial, es la situación en que nos encontramos con el siguiente programa, como capitanes de un poderoso navío el *H. M. S. RUN* de la *Royal Navy*.

Nuestro objetivo es el hundimiento del mayor número posible de submarinos enemigos. Para ello nos desplazaremos a izquierda y derecha por medio de las teclas 1 y 2, respectivamente, y lanzaremos las mortíferas cargas de profundidad pulsando la tecla 0.

Hay que tener muy en cuenta, que disponemos de un máximo de cuatro cargas cada vez, es decir, cuando se hayan lanzado cuatro cargas tendremos que esperar hasta que alguna de ellas explote, bien por alcanzar el submarino enemigo, o bien por reventar contra el fondo marino, antes de poder arrojar una nueva, las cargas serán lanzadas por la parte izquierda del buque, y podremos saber las disponibles en cada momento fijándonos en el centro de la fila superior de la pantalla.

Contamos con tres oportunidades antes que el Almirantazgo nos retire el mando a causa de nuestra falta de pericia, a modo de recordatorio, en la parte superior derecha de la pantalla, aparece una reproducción de nuestro buque por cada oportunidad que nos queda.

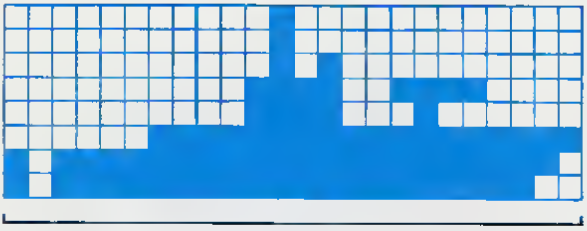
El submarino circula siempre de izquierda a derecha, y no se limita a dar paseos hasta que tengamos a bien sumergirle definitivamente, sino que intenta defenderse por diversos medios.

— surgiendo en cada ciclo a alturas diferentes y
— lanzando torpedos.

Si alguno de éstos llegara a alcanzarnos... ¡adiós oportunidad!, y si agotamos todos nuestros intentos, el programa finalizará pidiendo nuestra opinión sobre comenzar una nueva partida. Por otra parte, en la zona inferior de la pantalla (subsuelo marino), se señala permanentemente nuestra puntuación actual y la máxima puntuación obtenida en la jornada.

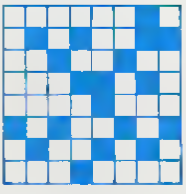
El juego utiliza la subrutina de caracteres gigantes de PSION, y por tanto, una vez grabado el programa BASIC definitivo, deberá grabarse a continuación una copia de dicha subrutina de código.

Las dimensiones del submarino son menores que las del acorazado, solo tres gráficos: E, F y G.



SUBMARINO

Cuatro son los gráficos que componen la silueta del acorazado: A, B, C y D.



EXPLOSION

ACORAZADO


A7

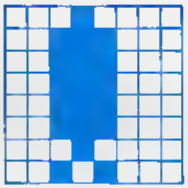
```
  1 REM * J.M  MAYORAL SERRANO *
 10 LET A$ "                                  ": LET B
$ "####" : LET C$ A$( TO 6)+ ABCD ABCD": LET N=0
 20 BORDER 0: LET A 0: LET B 3: POKE 23658,8
 30 PAPER 1: INK 1: CLS : PAPER 5: PRINT A$;A$;A$
 40 PAPER 1: INK 0: LET Z$=" PUNTOS:          RECORD
 *STR$ N+A$( TO 8-LEN STR$ N): LET XX=0: LET XY 193
LET XS=1: LET YS=1: GO SUB 650
 50 PAPER 1: INK 4: FOR I 0 TO 2: PLOT 0,I: DRAW 255
,0: NEXT I
 60 RESTORE 70: FOR I=1 TO 2: READ J: READ K: PLOT
J K: READ J: DRAW J,0: NEXT I
 70 DATA 0,3,40,44,3,25,95,3,12,125,3,12,160,3,42,21
0,3,45
 80 DATA 0,4,30,48,4,28,90,4,8,128,4,4,162,4,39,21,1
4,44
 90 DATA 0,5,37,45,5,10,99,5,4,129,5,1,163,5,37,212,
5,43
100 DATA 0,6,39,52,6,12,101,6,2,130,6,1,167,6,28,217
,6,39
110 DATA 0,7,20,55,7,6,173,7,16,227,7,28
120 DIM A(3,2): LET E=1: LET L=0
130 LET H=13: GO SUB 690
140 PAPER 4: INK 0: LET Z$=STR$ A: LET XX=64: LET XY
193: GO SUB 650
150 LET C=6+INT (RND*15): LET D=1
160 PAPER 5: INK 2: PRINT AT 0,16 C$( TO D+5) A$( TO
 8): AT 1,16;B$( TO 4 ):A$( TO 3)
170 LET D=D+1: IF D>2 THEN  PAPER 1: PRINT AT C 0;A
$ TO 3): GO TO 150
180 IF NOT ATTR (C,D+2) 1 THEN  GO TO 230
190 FOR I 1 TO 4
200 IF A(I,1)=C THEN  GO TO 220
210 NEXT I
220 LET D=D-1: GO SUB 690: GO TO 140
230 PAPER 1: INK 1: PRINT AT C,0,"  ": INK 2: "EEG
240 IF NOT E THEN  GO TO 270
250 LET E=INT (RND*C 4)  IF E THEN  GO TO 320
260 LET F=C 1: LET G=D+2
270 LET F=F 5: PRINT AT F+1 G    ;AT F+1 G+1,"
280 INK 6: PRINT AT F,G;"K";AT F,G+1;"K"
290 IF F>3 THEN  GO TO 320
300 PRINT AT F,G "  ";AT F,G+1;"  ": LET E=1
310 IF ATTR (2,G)=2 OR ATTR (2,G+1)=2 THEN  PRINT
AT 2,H+1, PAPER 5: INK 2 "JJJJ": GO SUB 590: PAPER 5
PRINT AT 2,H+1 A$( TO 4):AT C,D+1, PAPER 1;A$( TO 3)
 LET H=2 1: GO TO 30
320 FOR I=1 TO 4
330 IF NOT A(I,1) THEN  GO TO 390
340 IF A(I,1)=20 THEN  INK 2: PRINT AT A(I,1),A(I,2)
"J": GO SUB 590: PRINT AT A(I,1),A(I,2)  : LET A(I
,1)=0: LET L=L-1: GO TO 390
350 IF ATTR (A(I,1)+1,A(I,2))=15 THEN  GO SUB 680: G
O TO 190
360 LET A(I,1)=A(I,1)+1: INK 1: PRINT AT A(I,1) 1,A(
I,2)
370 LET M=0: IF INT (A(I,1)/2)-A(I,1)/2 THEN  LET M=
1
380 INK 3: PRINT AT A(I,1) A(I,2) ( H  AND M)+( I" A
ND NOT N)
390 NEXT I
400 LET X$=INKEY$: IF X$< "5" OR X$>"8" THEN  GO TO 5
10
410 IF X$="1" AND N THEN  LET N=N-1: GO TO 500
420 IF X$="2" AND H<26 THEN  LET H=H+1: GO TO 500
430 IF X$<>"0" THEN  GO TO 510
440 IF L=4 THEN  GO TO 510
450 LET L=L+1
460 FOR I=1 TO 4
470 IF A(I,1) THEN  GO TO 490
480 LET A(I,1)=3: LET A(I,2)=H: GO TO 510
490 NEXT I
500 GO SUB 640
510 FOR I=1 TO 40*10+L: NEXT I
520 IF B THEN  GO TO 580
530 PAPER 1: INK 7: FLASH 1: LET Z$="OTRO INTENTO (S
/N) ?": LET XX=48: LET XY=89: LET YS=2: GO SUB 650: F
LASH 0
540 LET X$=INKEY$: IF X$<>"S" AND X$<> N  THEN  GO T
O 540
550 IF X$ "N" THEN  GO TO 10000
560 IF A=N THEN  LET N A
570 GO TO 20
580 BEEP .08,50: GO TO 160
590 FOR J=-5 TO -15 STEP -1: BEEP .01,J: NEXT J: RET
URN
600 PRINT AT A(I,1),A(I,2)"
610 PAPER 1: INK 2: PRINT AT C D+1; JJJ : GO SUB 590
620 PRINT AT C D+1;A$( TO 3)
630 LET A(I,1)=0: LET L=L-1: LET A A+INT (C/(L+1))
RETURN
640 PAPER 5: INK 5: PRINT AT 2,H+1;"  ": INK 2:"ABCD"
INK 5;"  ": RETURN
650 LET I=23304: POKE I,XX: POKE I+1,XY: POKE I+2,8*
 POKE I+3,YS: POKE I+4,1: LET L=LEN Z$: FOR J=1 TO L
 POKE 23310+J,CODE Z$(J): NEXT J: POKE 23311+L 255-
LET L-USR 22256: RETURN
660 REM AUTOEJECUCION
670 CLEAR 32255
680 LOAD "LIT CODE
690 DATA 0,0,3,1,3,255,127,63
700 DATA 32,32,159,249,249,255,255,255
710 DATA 0,0,1,231,255,255,255,255
720 DATA 0,1,0,0,128,255,255,252
730 DATA 0,0,0,0,3,191,191
740 DATA 255,255,255,48,255,255,255
750 DATA 0,0,0,2,255,64,255,254,252
760 DATA 0,1,0,64,0,0,0,0
770 DATA 0,0,24,24,24,24,0,0
780 DATA 2,83,36,24,0,165,74,16
790 DATA 16,36,66,66,66,66,16,40
800 RESTORE 690
810 LET Q$ "ABCDEFGHIJK

820 FOR I I TO LEN Q$
830 FOR N=0 TO 7
840 READ A
850 POKE USR Q$(I)+N,A
860 NEXT N
870 NEXT I
880 RUN
```

máquina, que encontraremos en el programa LA BOMBA

La grabación del programa deberá efectuarse mediante el comando **SAVE "ACORAZADO" LINE 660,** aunque antes debemos recordar sustituir los caracteres que aparecen subrayados en el listado, por los gráficos de las teclas correspondientes, es decir, aquellos que se obtienen pulsando la letra subrayada cuando el cursor se encuentra en modo G **(CAPS SHIFT + 9).**

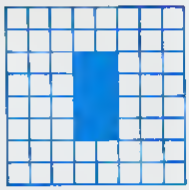
Los torpedos lanzados por el submarino son representados gracias al gráfico de la K



BOMBA

Mediante la utilización alternada de los gráficos H e I, se consigue el desplazamiento y giro de las cargas de profundidad.

CARGA HORIZONTAL



CARGA VERTICAL